Virtualization-based Techniques for Enabling Multi-tenant Management Tools

Chang-Hao Tsai[1], Yaoping Ruan[2], Sambit Sahu[2], Anees Shaikh[2], and Kang G. Shin[1]

[1] Real-Time Computing Laboratory, EECS Department
The University of Michigan, Ann Arbor, MI 48109-2121, USA
{chtsai,kgshin}@eecs.umich.edu,
[2] IBM TJ Watson Research Center, Yorktown Heights, NY 10598, USA
{yaopruan,sambits}@us.ibm.com,aashaikh@watson.ibm.com

Abstract. As service providers strive to improve the quality and efficiency of their IT (information technology) management services, the need to adopt a standard set of tools and processes becomes increasingly important. Deploying *multi-tenant* capable tools is a key part of this standardization, since a single instance can be used to manage multiple customer environments, and multi-tenant tools have the potential to significantly reduce service-delivery costs. However, most tools are not designed for multi-tenancy, and providing this support requires extensive re-design and re-implementation.
In this paper, we explore the use of virtualization technology to enable multi-tenancy for systems and network management tools with minimal, if any, changes to the tool software. We demonstrate our design techniques by creating a multi-tenant version of a widely-used open source network management system. We perform a number of detailed profiling experiments to measure the resource requirements in the virtual environments, and also compare the scalability of two multi-tenant realizations using different virtualization approaches. We show that our design can support roughly 20 customers with a single tool instance, and leads to a scalability increase of 60–90% over a traditional design in which each customer is assigned to a single virtual machine.

1 Introduction

As service providers look for new ways to achieve high quality and cost efficiency in the way they manage IT infrastructure for customers, an important emerging theme is the need to adopt a standard set of management tools and processes. This goal is complicated by the complex variety of customer environments and requirements, as well as the increasingly distributed nature of infrastructure management in which technical teams provide support for systems and networks located across the globe.

One recent strategy being pursued by IT service providers to address this challenge is to deploy a relatively small set of “best-of-breed” management tools that support *multi-tenancy*, *i.e.,* a single instance can support multiple customers. Multi-tenant tools have a number of important advantages in terms of cost and simplicity. They require deployment of a much smaller infrastructure, in contrast to having a dedicated installation for each customer, which can significantly reduce support costs for the infrastructure hosting the tool itself. Moreover, in some cases, multi-tenant tools have more advantageous software licensing models, for example, with a single license used to manage multiple customers. Finally, multi-tenant tools are a crucial element of the higher-level goal of consolidating tools to reduce training, management, and support costs.

A major barrier to adopting multi-tenant tools is that the desired management tool may not have been designed for multiple customer environments, and would thus require a significant rewrite to provide the needed support. Full multi-tenant support requires adequate, auditable, protection against the risk of data leakage between customers, performance that approaches the single-tenant case for each customer, and a relatively high density to realize the benefits of multi-tenancy.

In this paper, we explore the use of virtualization technology to enable multi-tenancy for systems and network management tools with minimal, if any, changes to the software itself. We consider virtualization at several layers, including full system virtualization, OS-level virtualization, and data virtualization. We describe and evaluate the trade-offs of these approaches through investigations of several design choices and experimental evaluations of their performance and scalability.

Our study focuses on OpenNMS, a popular open source, enterprise-grade network management tool that performs a number of functions including device discovery, service and performance monitoring, and event management [1]. We design multi-tenant-capable configurations of OpenNMS using the Xen virtual machine monitor (VMM) [2], and the OpenVZ virtual private server environment [3]. We then perform a number of detailed profiling experiments in which we measure the resource requirements and performance of OpenNMS in the virtual environments. These results also allow us to accurately configure Xen or OpenVZ in order to provide suitable performance. Finally, we compare the scalability of our virtualization-based designs with a baseline deployment in which each customer is assigned to a single Xen VM which houses the entire OpenNMS stack. We find that both Xen and OpenVZ can support multi-tenant deployments for nearly 20 customer networks (customer density with OpenVZ is slightly higher than with Xen), although each approach has its own relative advantages. Both, however, provide an overall scalability increase of 60–90% over the baseline configuration.

Many systems and network management tools adopt an architecture similar to OpenNMS, consisting of a web-based user interface application, a management server application which performs most of the monitoring or management functions, and a database that stores configuration information, collected data, and analysis reports. While our implementation considers multi-tenancy support in OpenNMS, we expect that the techniques and findings described in the paper will be applicable to a number of management tools that use this canonical architecture. Hence, our work represents an initial set of guidelines for leveraging virtualization technology to realize the increasingly important requirement to support multi-tenancy in systems management platforms.

The next section describes some background on network management systems. Section 3 illustrates our design choices in making the OpenNMS architecture multi-tenant-capable. We describe our testbed and experimental evaluation in Sect. 4. A brief discussion of related work appears in Sect. 5, and the paper concludes in Sect. 6.

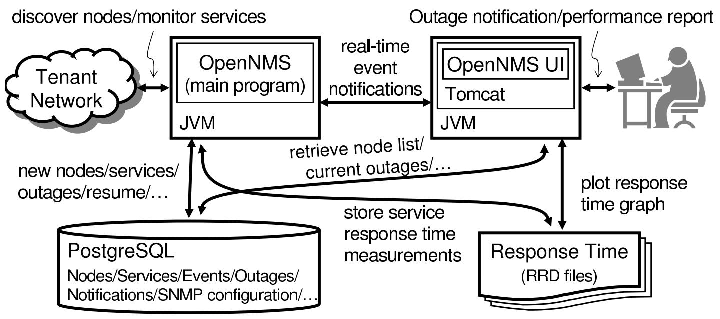
discover nodes/monitor services
Outage notification/performance report
Tenant Network
OpenNMS (main program)
JVM
real-time event notifications
OpenNMS UI
Tomcat
JVM
retrieve node list/ current outages/…
new nodes/services/ outages/resume/…
plot response time graph
store service response time measurements
PostgreSQL
Nodes/Services/Events/Outages/ Notifications/SNMP configuration/…
Response Time (RRD files)

Fig. 1. OpenNMS architecture

2 Background

Network management is a standard service in current enterprise environments. These systems are a combination of hardware and software that monitors the topology, availability, and traffic of various network devices like routers and switches, as well as servers that provide services on the network.

While network management tools can be provided as a service, they differ from other services because they must connect to customer networks through firewalls at their network edge and customers may use network address translation (NAT) to employ private Internet addressing in the local network. For multi-tenant management tools, this presents two challenges. First, the tool cannot be deployed within one customer network because it needs to monitor multiple customer network domains and these private addresses are not publicly accessible. Second, private addresses may cause confusion to the tool because of overlapping addresses between customers.

We use a popular open-source NMS, OpenNMS, as our target application. OpenNMS is billed as an enterprise-grade network management platform, and is used in a variety of commercial and non-commercial environments [1]. OpenNMS monitors network-service availability, generates performance reports, and provides asset-management capability. Figure 1 shows major components of OpenNMS and their interactions.

The management server software is implemented in Java as a multi-threaded application. We call this part the *back-end*. The *front-end* user interface (UI) consists of a number of servlets and Java server pages (JSPs) deployed in an Apache Tomcat application server. Both the front-end and back-end connect to a PostgreSQL database for various management information. Response time of network services and SNMP counters are stored in Round Robin Database (RRD) files and later plotted in the user interface. Besides notifying users via the UI, OpenNMS can also be integrated with email or instant messaging systems to send timely notifications of critical events.

3 Design

In this section, we describe the design of our multi-tenant-capable OpenNMS using virtualization with minimum changes to the original system. With virtualization tech-

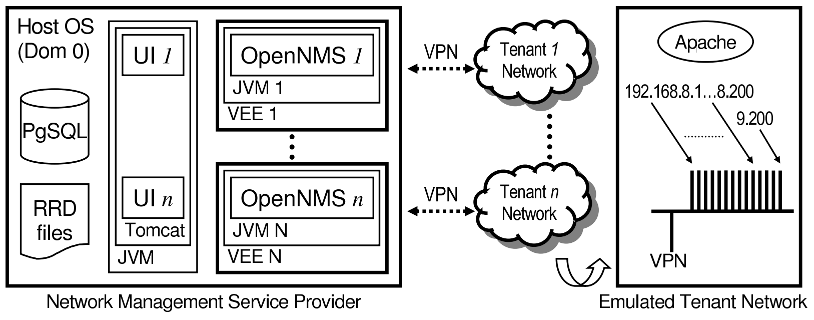

Fig. 2. Design overview of the multi-tenant-capable OpenNMS

nology, the brute-force solution is to select a virtualization platform and run one OpenNMS tenant in each virtual execution environment (VEE), including all components described in Fig. 1. However, this solution is not efficient — as shown in our evaluation, memory becomes the bottleneck of the system. We propose an approach to virtualize only the back-end, and share a common database among all tenants.

An overview of this design is given in Fig. 2, where the OpenNMS back-end of each tenant sits in an VEE, and communicates with its managed network domain via VPN connections. The database and Tomcat are co-located in one VEE. Database queries from the back-ends are configured to use the tenant ID in the database name. All of these components can be organized together by configuration changes, to meet our goal of no modification to the source code.

3.1 Virtualization for the Back-end

The back-end of OpenNMS is the core of the system. It loads configuration files when it starts and instantiates internal service modules accordingly. The configuration files usually include customer-specific parameters such as the network address range for automatic service discovery, service-polling frequency, *etc.* Once the configurations are loaded, it starts probing of the customer network domain for each service accordingly.

We choose a virtualization implementation that provides low overhead but necessary isolation required by OpenNMS to work properly. First, it should provide file system virtualization so persistent states of each tenant such as configuration files are protected. Second, it should provide process and memory isolation so tenants on the same platform do not interfere with each other. Finally, since each tenant needs to communicate with its own network domain, the network layer should be virtualized as well. Especially when two tenants have identical private network addresses, packets from each tenant should be routed correctly. This requirement implies that each host should maintain its own protocol stack. For these reasons, virtualization technologies such as Java virtual machine (JVM) and FreeBSD Jail are not sufficient.

We use Xen and OpenVZ as our virtualization platforms in our implementation. Both of them provide virtualized network protocol stacks. We create multiple VEEs within a host and run an instance of OpenNMS in each VEE. Since each VEE also provides a resource boundary, performance isolation can also be implemented. We measure the performance of each tenant and identify the location where user-perceived performance is degraded.

3.2 VPN Connections

Traditionally, a management system is deployed inside each customer's network domain. With multi-tenancy, the system has to be located at a place where all customers can reach. However, most enterprise networks are not reachable from outside. Changing firewall configuration at each customer's network edge to accommodate this communication may introduce potential security risks. One approach to solve this problem is to use probing devices within each tenant's premise. While this approach might be feasible in real deployment, we choose to create Virtual Private Network (VPN) connections to each tenant's network. Creating VPNs is better for portability than probing devices, and is easier for setting up an experimental testbed. In this paper, we use OpenVPN [4], an open source SSL VPN solution. We configure OpenVPN to establish layer 2 (L2) VPN connections to tenant networks to support services such as BOOTP and DHCP servers, although most of the services can be monitored via a layer 3 (L3) VPN. When there are multiple L2 networks to be monitored, several VPN connections can be established simultaneously.

Using VPN connections to connect NMSs to tenant networks does incur some overhead in packet transmission due mainly to round-trip time, which depends on network congestion and the geographical distance between VPN endpoints. However, for management systems, this delay makes little impact on the results. For example, when monitoring service-availability, the added overhead does not pose any problem as long as the probing returns without timeout, which is 3 seconds.

3.3 Database Sharing

Since each OpenNMS instance only needs a database user account to store data in a PostgreSQL database server, we opt to use mechanisms built in database to provide the isolation for each tenant.

We configure each tenant to use a different database user name and database instance to store the data. This approach provides adequate security isolation, since data belongs to one tenant is not accessible by the others. As far as performance is concerned, database access usually is not the bottleneck in a multi-tenancy environment. High-availability database design can be used to prevent any database crash. We do not use these designs so as to compare results with the brute-force solution.

3.4 The Front-end

Similar to consolidating databases, we deploy multiple instances of the Java servlets and JSPs in Apache Tomcat. This allows customizing the front-end user interface to fit each customer's management policy and preferences. Each tenant has a different base URL to its web console. In addition to log-in credentials, access control can be applied to restrict access further.

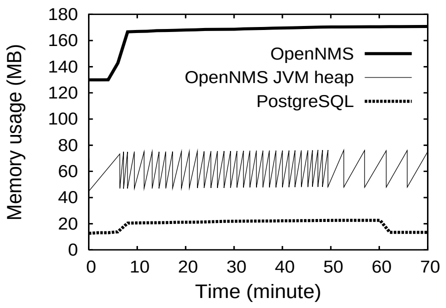

Fig. 3. Memory-usage profile of OpenNMS, PostgreSQL and JVM heap

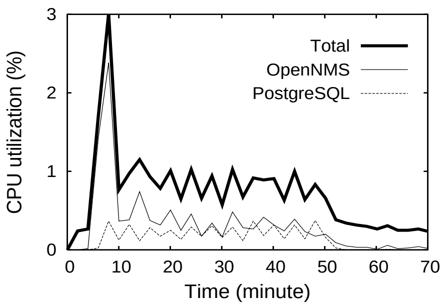

Fig. 4. CPU-utilization profile of the entire system, OpenNMS, and PostgreSQL

4 Evaluation

In this section we describe our experimental evaluation which comprises two sets of experiments. The first set of experiments profiles and determines the working set and resource bottlenecks of our setup. The key results are: (i) memory is the bottlenecked resource, and (ii) the minimum memory required for each OpenNMS setup is about 144MB. The second set of experiments compares the benefit of our multi-tenancy approach with that of the baseline approach that is void of any multi-tenancy capabilities. We find that even with our "limited multi-tenancy" approach, as many as 60–90% more tenants can be supported with similar or better perceived quality.

4.1 Testbed Setup

We use two similarly-equipped machines to emulate the management server and the tenant networks. Each of them has an Intel Core 2 Duo E6600 CPU (without using Intel Virtualization Technology), 4GB of main memory, and two 7200rpm hard drives. The management station and the emulated tenant network are connected with a Gigabit Ethernet network. Debian GNU/Linux 4.0 is used as the host OS environment with PostgreSQL database server and Apache Tomcat 5.0. The OpenNMS we used in this study is version 1.2.9. Sun Java 5.0 is used to compile and run OpenNMS.

Our Xen-based implementation uses an experimental version of Xen, including a modified Linux kernel 2.6.18 as the kernel in both privileged and unprivileged domains. We optimize the setup by having the tenants share a common NFS-mounted `/usr` since all of them use the same program and do not need to contain identical files in their virtual disk images. Sharing file systems also improves the cache-hit ratio in the host OS. Another approach to reducing the file system size is to use copy-on-write disks. Unfortunately, this feature is not stable in our testing. The result is a 150MB root file system for each tenant and a 350MB `/usr` file system shared by all tenants. We also give each guest OS a 256MB swap space.

For the testbed using OpenVZ, we use a patch set (version 028stab027.1) for Linux kernel 2.6.18 in this work. A VEE created by OpenVZ shares the same patched Linux kernel but has its own root file system, network device, *etc.* We configure the software installation identically as in Xen.

In order to test our design, we emulate a tenant network as shown in Fig. 2. All tenants share the same emulated tenant network, which is created on a dummy network interface configured with 1,000 IP addresses. An Apache HTTP server listens on all IP addresses to create an illusion of 1,000 web servers. System parameters, such as increasing buffer size, are tuned to make sure that network and the client machine are not the bottlenecks.

4.2 Resource Profiling

The intent of this evaluation is to profile the resource usage of our proposed multi-tenancy-capable network management tooling using OpenNMS and provides parameters to use in multi-tenant evaluations. We determine resource bottlenecks, the working set for the proposed setup, and any trends as the number of clients being monitored are scaled. The first three parts of resource profiling are ran without any VMM.

Memory is the Bottlenecked Resource. We first ran OpenNMS with the database server within one OS, just as the typical setup for single tenant system. The OpenNMS is set up to use at most 256MB memory as JVM heap and monitor 200 hosts via a VPN connection.

Figure 3 presents the system memory used by OpenNMS, PostgreSQL and OpenNMS JVM heap usage as a function of time progression. When OpenNMS starts up, it first loads its configuration and previously-monitored hosts and services (none in this evaluation) from the database. It starts by discovering the clients to be monitored 5 minutes after the boot-up. During this stage, although the heap utilization is increased, memory used by the OpenNMS remains flat. Once auto-discovery starts, OpenNMS uses considerably more memory and the garbage collection of JVM kicks in periodically, generating a zig-zag shape of heap utilization between 49MB and 78MB. The increase in memory usage by the OpenNMS can be attributed to dynamic class loading and objects in the permanent generation, which is not included in heap utilization.

The auto-discovery procedure is paced by OpenNMS to avoid generating too much traffic in the network. Therefore, the duration of this stage is proportional to the number of probes and the number of hosts being monitored. Using the default configuration, it takes about 45 minutes to run all probes over 200 hosts. Since the emulated client network has only one Apache HTTP server running, most of the time is spent on waiting timeouts. Both OpenNMS and PostgreSQL use more memory as the auto-discovery procedure goes on.

After the auto-discovery completes, OpenNMS only periodically probes previously-discovered and manually-configured hosts and services, and thus, creates new Java objects at a slower rate, which leads to less-frequent garbage collection. PostgreSQL server also frees some memory as most data, such as event logs of each host, are not actively being used. The VPN connection, OpenVPN, uses 2.5MB memory constantly which is not plotted in the figure.

In terms of CPU utilization, which is plotted in Fig. 4 shows the CPU utilization as a function of progression of time. Note that the peak utilization occurs when the

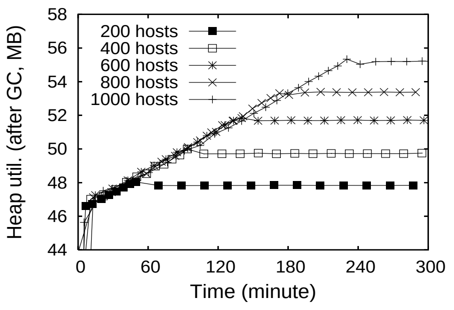

Fig. 5. Heap utilization vs. client network size

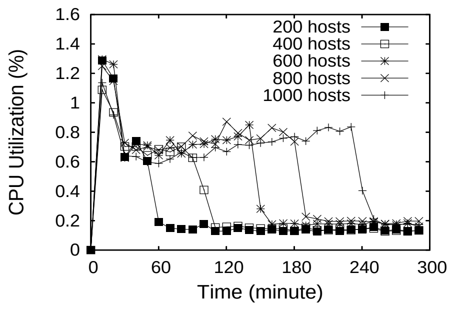

Fig. 6. CPU utilization vs. client network size

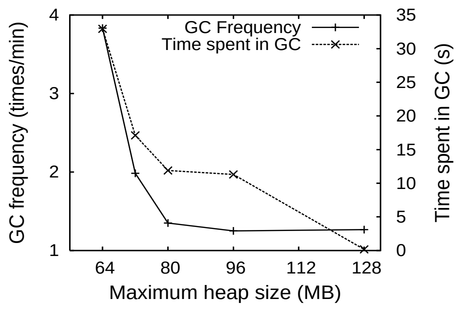

Fig. 7. GC frequency and time vs. heap size

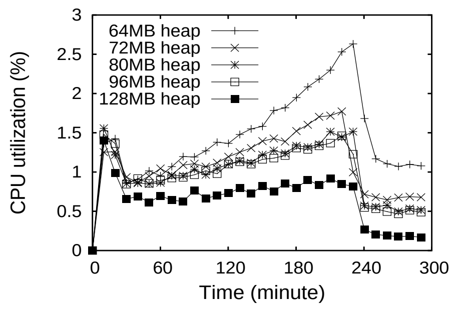

Fig. 8. CPU utilization vs. heap size

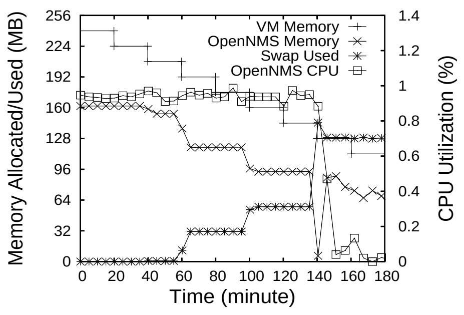

Fig. 9. Swap activity vs. VM memory size

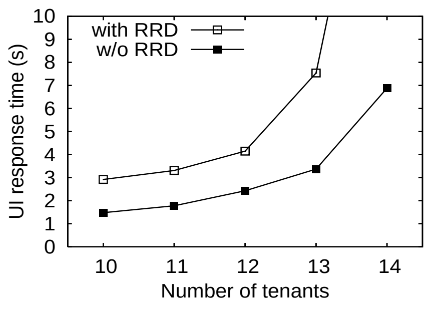

Fig. 10. Scalability of the baseline multi-tenancy

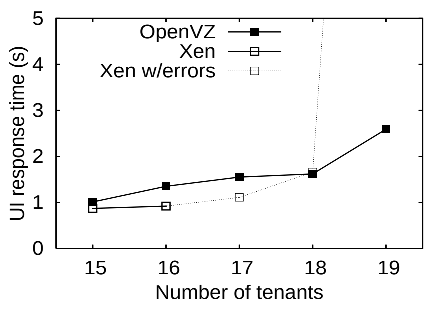

Fig. 11. Front-end UI response time with different number of tenants

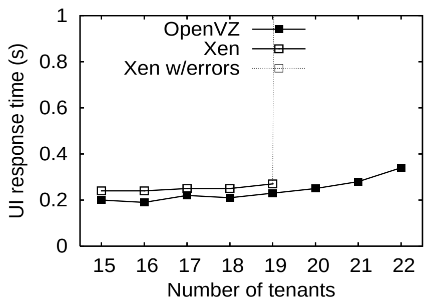

Fig. 12. Front-end UI response time when response time logging is disabled

auto-discovery phase starts. The CPU utilization then stays around 1% during the auto-discovery stage and drops to almost 0 afterward, where the OS overhead is around 0.3%. From these two figures, we conclude that system memory is the potential bottleneck when multiple instances of OpenNMS are hosted on the same machine.

Effect of Client Network Size. We then study the impact of the client network size on the resource utilization by varying the client size from 200 hosts to 1,000 hosts, *i.e.,* the typical network size in small to medium size businesses. The maximum JVM heap size is also 256MB as in the previous experiment.

While the memory used by OpenNMS does not differ much with different client network sizes, we observed that the heap utilization, after each garbage collection, is proportional to the number of hosts being monitored. From Fig. 5 we observe that for every 200 monitored hosts, OpenNMS uses 2MB of additional memory in heap. Comparing to the size of the OpenNMS process, this incremental cost is low.

Figure 6 shows that CPU utilization is only slightly affected by the client network size. This result reinforces our previous observation that system memory size is the bottleneck for OpenNMS when multi-tenancy is enabled.

Effect of JVM Heap Size. Next, we evaluate the effect of JVM heap size on our proposed multi-tenant-capable network management tool. We configure OpenNMS to monitor a client network consisting 1,000 hosts, and reduce the maximum heap size from the default 256MB to 64MB to investigate their relationship.

As we can see in Fig. 7, the garbage collection frequency is inversely related to the maximum heap size. The frequency is measured after auto-discovery is completed. When the maximum heap size is reduced to 64MB, garbage collection happens as frequent as 4 times a minute. In spite of this frequent garbage collection, the total time spent in the garbage collection in an hour is as little as 33 seconds, which is less than 1% CPU utilization. In addition, we observe that the JVM spent much less time in garbage collection if there are 128MB or more heap space, where a different garbage collection algorithm might be used. We also tried a 56MB heap configuration but the JVM could not survive.

The increase in CPU utilization is more pronounced in auto-discovery stage as can be seen in Fig. 8. With as little as 64MB heap size, OpenNMS uses as much as 2.5% CPU time at the end of auto-discovery stage. The increase in the CPU utilization with the number of host discovered suggests the garbage collector needs more time to sweep out dead objects among an increasing number of alive ones, and this phenomenon is more obvious when the heap size is smaller.

Working Set Estimation. Next, we determine the memory working set size for OpenNMS. While OpenNMS takes up as much as 160MB of memory to run, like most applications, the working set size is usually much smaller than the size of the total virtual memory segments that reside in physical memory. In the following, we take advantage of Xen's dynamic memory resizing capability and reduce the memory allocation of a VM from 256MB down to 96MB (at a rate of 16MB/20min), and monitor the swap space usage. In this experiment, only OpenNMS and OpenVPN are running in a VM, PostgreSQL has been moved to domain 0 as the database server will be shared between multiple instances of OpenNMS.

In Fig. 9, we observe that the dirty memory pages begin to be swapped out to the swap partition when physical memory is reduced to 192MB. Swap space usage increases again when VM is reduced further by 32MB. When only 128MB is allocated, Linux suddenly swaps out all the memory used by OpenNMS. Although the working set was reloaded immediately, the dramatic drop in CPU utilization implies that most of the time were spent in dealing with page faults. Therefore, we conclude that OpenNMS with OpenVPN requires at least 144MB to perform smoothly.

4.3 Evaluation of Multi-tenancy Benefits

We evaluate the number of tenants that can be supported for both baseline multi-tenancy and our proposed multi-tenancy capability. The metric used for this evaluation is the increase in the number of tenants that can be supported by the same amount of resources while providing similar or better quality of service compared to the baseline multi-tenancy capability. The quality of service metric is the UI response time in the measurement process, and correctness of discovery and availability results.

For this evaluation, we configure the testbed so that each tenant has 400 emulated clients to be monitored. All the instances are started simultaneously, thus it can be considered as the worst-case scenario. We wait 2 hours for the auto-discovery process to complete and start polling results from the UI. For each tenant, we first log-in to the web console, list the number of hosts being monitored, and randomly pick 10 hosts to list their details. We report the average response time for the UI operations where the average is computed across all the tenants over all the clients.

Scalability of Baseline Multi-tenancy. We first evaluate the scalability of a baseline multi-tenant OpenNMS installation, where each instance not only includes the back-end and OpenVPN but also the database and the Apache Tomcat server on top of a dedicated OS. Each tenant is hosted in a Xen VM with 256MB memory. Figure 10 shows that the UI response time increases with the number of tenants hosted. Although 14 tenants can be hosted on one server and discover all hosts and services, the UI response is an awfully 22s, which is completely unusable. If we set a response time threshold of 3s, only 10 tenants can be hosted.

While the bottleneck is the main memory size, the performance of the system can be improved by eliminating the disk activities resulting from keeping response time log files (RRD files). The UI becomes much more responsive and, as a result, the scalability improves to 12 tenants. However, we were not able to start 15 tenants due to out-of-memory errors.

Proposed Multi-tenancy Scalability. We then evaluate the scalability of our proposed multi-tenancy solution, where the database and the Apache Tomcat server are shared among all the tenants. Figure 11 shows that the average response time is significantly reduced. Also note that 16 and 19 tenants can be hosted when Xen and OpenVZ based virtualization is used, respectively. Comparing to baseline multi-tenancy approach, our proposed multi-tenancy solution can support as much as 60–90% more number of tenants with similar or better UI response time.

When 17 tenants are hosted using Xen virtualization, we have observed some transient outages while the emulated network did not undergo any failure. When the number of tenants increased to 18 or more, there are many hosts that were not discovered and lots of false alarms. We considered these cases failed to meet the standard and plotted with dashed lines.

On the other hand, OpenVZ is able to host 19 tenants without any failure but failed to run with 20 tenants. The average response time is higher than that in Xen because Apache Tomcat and the database server also need to compete for main memory with OpenNMS JVMs — in Xen OpenNMS JVMs are confined in their own domains.

When more than 19 tenants are hosted, we observe heavy disk activities from reading and writing RRD files. In stead of optimizing disk performance, we evaluate the scalability again without the response time logging. The results are plotted in Fig. 12. The response time is reduced significantly again for both Xen and OpenVZ. Xen and OpenVZ can host 19 and 22 tenants respectively without any false alarms. When hosting more tenants, memory becomes bottleneck again and causes errors. Compared to the baseline multi-tenancy model, we observe 58–83% scalability improvements, while providing much better response time.

5 Related Work

The concept of multi-tenancy is usually applied to enterprise software such as ERP and CRM. It reduces the cost of operating a software application by sharing the associated hardware and software licensing cost with other customers. Successful multi-tenant-capable applications are usually designed from the ground-up [5]. In this work, we apply multi-tenancy to a specific kind of application, systems and network management, using virtualization as the enabler. Comparing to other applications, network management cannot live in application layer alone. It interacts with customers' network infrastructure and must deal with facts like IP address conflicts between customers.

One approach to handle IP address conflicts is to use network address translation (NAT) to map conflicting addresses into non-overlapped addresses in network management service provider's network. This approach is proposed with management payload address translation (MPAT) to deal with IP addresses in SNMP payload by Raz and Sugla in [6]. While it enables servicing multiple tenants with one network management software installation, this scheme cannot deal with unstructured use of IP addresses in protocols such as command line interface (CLI) of various network devices.

The overhead of virtualization has been evaluated by several researchers[7, 8]. In particular, using Xen incurs some overhead in disk and network I/O and Linux-VServer, which is another OS-level virtualization mechanism and performs closely to native OS performance. As our evaluation result shows, the bottleneck of our testbed is either the amount of main memory or in the disk sub-system. Neither of them results from the use of virtualization. Implementing anticipatory scheduling in a VMM with guest context awareness as in [9] may improve disk throughput.

The memory footprint of each Xen VM is fixed in our implementation. Workload characterization helps us determine the optimal setting. Another approach to control memory allocation is to monitor its actual usage on-line [10, 11]. Unfortunately, JVM heap size cannot be changed accordingly at run-time. Without increasing JVM heap size with VM memory size, JVM cannot benefit much from additional memory. On the other hand, reducing VM memory allocation alone can lead to unnecessary swapping of dead objects.

6 Conclusion

In this paper we have described an approach to enabling multi-tenant capability in one of the popular network management tools, OpenNMS. We study the architecture of the management tool, and divide the system into different components including front-end, back-end engine, and storage database. We use virtualization as the base platform to ensure the isolation between different tenants. One single database is shared between multiple tenants to reduce the cost of hosting database servers and improve scalability. Our implementation using Xen and OpenVZ virtualization technology shows that both systems meet the requirements of multi-tenancy, and are able to provide about 20 tenants without reducing service quality.

References

1. OpenNMS Group: OpenNMS. `http://www.opennms.com`
2. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP'03). (2003) 164–177
3. OpenVZ Group: OpenVZ. `http://www.openvz.org`
4. OpenVPN Project: OpenVPN. `http://www.openvpn.net`
5. Fisher, S.: Service computing: The appexchange platform. In: 2006 IEEE International Conference on Services Computing (SCC '06). (Sept. 2006) xxiv (Keynote).
6. Raz, D., Sugla, B.: Economically managing multiple private data networks. In: 2000 IEEE/IFIP Network Operations and Management Symposium (NOMS '00). (2000) 491–503
7. Menon, A., Santos, J.R., Turner, Y., Janakiraman, G.J., Zwaenepoel, W.: Diagnosing performance overheads in the xen virtual machine environment. In: 1st ACM/USENIX International Conference on Virtual Execution Environments (VEE '05). (2005) 13–23
8. Soltesz, S., Herbert-Pötzl, Fiuczynski, M.E., Bavier, A., Peterson, L.: Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In: 2006 EuroSys Conference (EuroSys '06). (2006)
9. Jones, S.T., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Antfarm: Tracking processes in a virtual machine environment. In: 2006 USENIX Annual Technical Conference (USENIX '06). (June 2006) 1–14
10. Waldspurger, C.A.: Memory resource management in vmware esx server. SIGOPS Operating Systems Review **36** (2002) 181–194
11. Jones, S.T., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Geiger: Monitoring the buffer cache in a virtual machine environment. In: The 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII). (2006) 14–24